

CBLOCK: An Automatic Blocking Mechanism for Large-Scale De-duplication Tasks

Anish Das Sarma ^{#1}, Ankur Jain ^{*2}, Ashwin Machanavajjhala ^{*2}, Philip Bohannon ^{*2}

^{*1}Google Research, ^{*2}Yahoo! Research

{anish.dassarma}@gmail.com, {ankurja,mvna,plb}@yahoo-inc.com

Abstract

De-duplication—identification of distinct records referring to the same real-world entity—is a well-known challenge in data integration. Since very large datasets prohibit the comparison of every pair of records, *blocking* has been identified as a technique of dividing the dataset for pairwise comparisons, thereby trading off *recall* of identified duplicates for *efficiency*. Traditional de-duplication tasks, while challenging, typically involved a fixed schema such as Census data or medical records. However, with the presence of large, diverse sets of structured data on the web and the need to organize it effectively on content portals, de-duplication systems need to scale in a new dimension to handle a large number of schemas, tasks and data sets, while handling ever larger problem sizes. In addition, when working in a map-reduce framework it is important that canopy formation be implemented as a *hash function*, making the canopy design problem more challenging. We present CBLOCK, a system that addresses these challenges.

CBLOCK learns hash functions automatically from attribute domains and a labeled dataset consisting of duplicates. Subsequently, CBLOCK expresses blocking functions using a hierarchical tree structure composed of atomic hash functions. The application may guide the automated blocking process based on architectural constraints, such as by specifying a maximum size of each block (based on memory requirements), impose disjointness of blocks (in a grid environment), or specify a particular objective function trading off recall for efficiency. As a post-processing step to automatically generated blocks, CBLOCK *rolls-up* smaller blocks to increase recall. We present experimental results on two

large-scale de-duplication datasets at Yahoo!—consisting of over 140K movies and 40K restaurants respectively—and demonstrate the utility of CBLOCK.

1 Introduction

Integrating data from multiple sources containing overlapping information invariably leads to *duplicates* in the data, arising due to different sources representing the same entities (or facts) in slightly different ways; e.g., one source says “George Timothy Clooney” and another says “G. Clooney”. The problem of identifying different records referring to the same real-world entities is known as *de-duplication*¹. De-duplication has been identified as an important problem in data integration, and has enjoyed significant research interest, e.g. [11, 12, 13, 24, 27, 29, 33].

Conceptually, de-duplication may be performed by considering each pair of records, and applying some *matching function* [12, 21, 31] to compute a similarity score, then determining duplicate sets of records based on clustering similar pairs. However, comparing all pairs of records to be de-duplicated is prohibitively expensive in commercial or web applications that require matching data sets with millions of records (e.g., persons, business listings, etc). *Blocking* or *canopy-formation* (e.g., [4, 7, 15, 17, 19, 22, 23, 32]) has been identified as a standard technique for scaling de-duplication: The basic idea is to find a set of (possibly overlapping) subsets of the entire dataset (called *blocks*), and then compute similarity scores only for pairs of entities appearing

¹De-duplication is also known by many other names such as reference reconciliation, record linkage, and entity resolution.

in the same block. We use the term “blocking function” to refer to any function that maps entities to block numbers, usually based on the value of one or more attributes. One example of a blocking function would be the value of the “phone number” attribute, or the first seven digits of the same, etc. In an ideal situation, all (or most) of the duplicates would appear together in at least one block.

As a result, a good blocking function must be designed for *each large-scale matching task*. We are seeking to build a scalable system for de-duplication of web data. The system will be used for a wide variety of de-duplication tasks, and must support *agility*, the ability to rapidly develop new de-duplication applications. Accordingly, an important part of developing this system is effective, automatic construction of blocking functions. Like [30], de-duplication tasks in our system execute in a map-reduce framework like Hadoop. In this setting, computation is broken into rounds consisting of a *map phase* in which a set of keys is generated by which work is split over a potentially large number of compute nodes and a *reduce phase* in which partial results from each compute node are combined. A natural approach for de-duplication is to use the map phase to execute the blocking function, allowing match scores to be computed in parallel on each mapper.

In order to design appropriate blocking functions for our setting, we face four important challenges. First, a premium is placed on minimizing the number of rounds of computation in a map-reduce setting, since each round involves significant scheduling and co-ordination overheads. Second, data in our system comes from a variety of feeds, and is often noisy. In particular, attributes may be only partially populated, leading to asymmetric block sizes if these attributes are used for blocking. Third, matching is executed in parallel, meaning that a premium is placed on minimizing the size of the largest block without exceeding the maximum number of compute nodes available. Fourth, the complexity of the de-duplication process can be significantly reduced if every object is given only a single hash value for mapping; which we refer to as the *disjoint* blocking condition.

We present CBLOCK, a system that automatically creates canopies based on the information specified by the application. We now describe the approach taken in CBLOCK to address the above challenges. We introduce a conditional tree of blocking functions, the *BlkTree*. In this tree, blocks with large expected size are explicitly

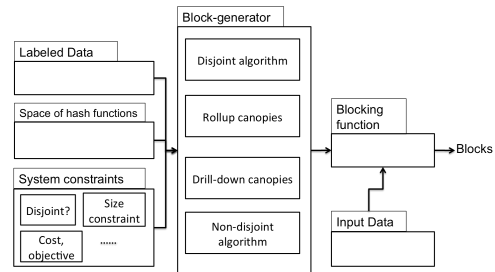


Figure 1: Components of the CBLOCK system

mapped to a child blocking function, making each path in the tree equivalent to a conjunctive blocking function applied to a subset of the data. The introduction of the Blk-Tree allows for an expressive blocking function, which allows us to effectively block even skewed data, such as attributes with many null values.

Second, to handle the situation in which the number of blocks exceeds the number of compute nodes, we introduce a *roll-up* step for the BlkTree to efficiently reduce the number of compute nodes without excessively increasing complexity of the hash function. Third, we optimize for the best blocking function while keeping the size of the largest block within a constrained size. Since the overall latency of the parallel computation corresponds to the slowest node, this is a natural optimization goal, but is not addressed by existing techniques. As an aside, we note that our system can also be used for other applications that require a similar capability as blocking: (a) In a binary classifier with many features, CBLOCK may be used to pick a small set of features that most effectively captures the classification; (b) We can use CBLOCK to determine which sets of values from two relations may contribute most to join results in a distributed join solution such as [25].

The flow of data through CBLOCK is illustrated in Figure 1. As input to the system, shown on the left, is a set of training examples consisting of true-positive match pairs shown at the top, and a set of configuration parameters shown at the bottom including size constraints, disjointness conditions and any tuning of the cost objective. These inputs feed into the CBLOCK system, shown in the middle block, that designs a blocking configuration. This configuration is then passed to the runtime system (e.g. a map-reduce system) for execution of the blocking as the first phase of the de-duplication algorithm.

1.1 Contributions

Our paper makes the following contributions, addressing the requirements of automatic blocking configuration for web-scale de-duplication:

- In order to decrease the number of rounds (disjuncts), we argue that it is necessary to increase the power of individual hash functions while still respecting disjointness constraints. In Section 4 we formally introduce *blocking trees* to accomplish this goal. We show that in general finding blocking trees that maximize recall subject to a *maximum block-size* constraint is NP-hard, and we provide a natural greedy algorithm.
- We show that adapting the state of the art solution of [7, 23] to optimize for the *maximum size* constraint can naturally be expressed as a special case of finding optimal blocking trees.
- Section 5 introduces the roll-up problem of merging small canopies produced using any disjoint blocking scheme. We show a close connection of the roll-up problem with the knapsack problem, establish the NP-completeness of solving the general problem, and provide a heuristic algorithm based on a 2-approximate algorithm for the knapsack problem.
- Section 6 studies “drill-down” problem, i.e., given a domain of an attribute and a labeled dataset of true duplicates we want to find optimal hash functions that meet a canopy-size requirement. We formally define the problem and present a near-linear time optimal algorithm based on dynamic programming.
- For most of the paper we focus our attention on disjoint blocking functions. Section 7 extends our study to non-disjoint blocking functions. To our knowledge, this is the first work to consider the disjointness issue for blocking design.
- CBLOCK is fully implemented along with all the functionality described above. In Section 8, we describe our system and present experimental results on two large commercial datasets consisting of around 140K movies and 40K restaurants respectively.

Related work is described in Section 2. Due to space constraints, formal proofs for all technical results are omitted from the paper.

2 Related Work

To the best of our knowledge, ours is the first work to: (1) Present techniques on finding blocking functions by explicitly trading-off recall for efficiency, and in a more expressive tree-based structure than flat conjunctive structures of past work; (2) Formally introduce and study the problem of *rollup* as an important post-processing step to assemble small canopies and increase recall; (3) Provide automatic solutions to the *drill-down* problem as a way of bootstrapping blocking with no manual effort, or augmenting manually-generated hash functions; (4) Present an automatic blocking system for de-duplication in a distributed setting that is applied to two large commercial datasets from a search engine. Very few pieces of previous work consider blocking based on labeled training data, while there is a much larger body of work on hand-tuned blocking techniques using similarity functions. We start by describing the relationship of our work with blocking based on labeled data (Section 2.1), followed by blocking without labeled data (Section 2.2), and finally other work on de-duplication (Section 2.3).

2.1 Blocking With Labeled Data

Two recent papers [7, 23] presented approaches to constructing a blocking function using a labeled dataset of positive and negative examples. Roughly speaking, both papers learn conjunctive rules (and disjunctions of conjunctive rules) to maximize recall. [7] attempts to maximize the number of positive minus negative examples covered, effectively using negative examples as a proxy for minimizing the size. [23] uses only positive examples, but does not explicitly incorporate any size restriction. Below we give a detailed comparison with these past approaches:

- We present BlkTrees, a more expressive language for expressing disjoint blocking functions than previous work. Given only simple or conjunctive blocking functions, it may not be possible to construct an effective blocking function without a large number of map-reduce rounds (disjuncts).
- Minimizing negative training examples covered by a blocking solution may lead to quality problems from *overly aggressive blocking*. For example, consider a movie and a remake with the same title but released in a different year – while the two are a negative exam-

ple, this does not mean it is a bad idea to block movies together when their titles are similar. In short, blocking should be optimized for *recall* vs. *efficiency*, and match rules optimized for *precision*.

- Minimizing negative training examples does not match the cost model of *parallel computation* models like map-reduce, where latency is determined by the *largest block*.
- We are the first to introduce and solve the rollup and drill down problems. These problems were not addressed in [7, 23], or in other past work on blocking.

2.2 Blocking Without Labeled Data

[15] introduced the notion of blocking (called “merge-purge”) by constructing a key for each record, sorting based on the key, and then performing matching and merging in a sliding window. [15] (and other variants [17, 19]) do not consider automatic generation of optimal blocking functions in a distributed environment, based on training data.

SWOOSH [5] is a recently developed generic entity resolution system from Stanford. Their specific paper on blocking [32] focuses on “inter-block communication”, by propagating matched records to other blocks. Once again, automatic generation of blocking functions is not the subject of [32]. Further, D-Swoosh [6] (and other similar work [25, 28]), their distributed framework for entity resolution focus on distributing pairwise comparisons across multiple processors, as opposed to our focus of partitioning the data to reduce the number of total pairwise comparisons.

Reference [22] presented techniques for generating non-disjoint canopies based on distance measures such as jaccard similarity of tokens. After choosing a distance function, they pick records as canopy centers, and add to each canopy all records that are within some distance based on the distance measure. The algorithms from [22] cannot be directly scaled to a distributed environment. A similar approach of generating (non-disjoint) canopies by clustering based on any distance measure was also proposed in [26]. Some other work [9] considers blocking based on bi-grams of string attributes, followed by creation of inverted lists for each bigram. Another recent piece of work [18] considered transforming the data into a euclidean space. While the above approaches weren’t designed specifically for a distributed environment, re-

cently [30] studied the problem of performing approximate set similarity joins using a map-reduce framework. Their work can be used for blocking when records are compared for duplicates based on set similarity functions. Also, a recent system, MAHOUT [3], described an implementation of canopy clustering in a map-reduce framework. Finally, [4] performed a comparative study of blocking strategies from [9, 15, 17, 22].

In general, the approaches described above rely on the knowledge of specific similarity/distance functions. Furthermore, they necessarily generate non-disjoint canopies, whereas one primary goal of our work was to consider disjoint canopies as an important choice for distributed de-duplication and obtain non-disjointness as multiple rounds of disjoint sets of canopies. Finally, none of this past work considers the rollup and drill-down problems.

2.3 Other Work

De-duplication has been studied for over 50 years now, starting with the seminal pieces of work in [12, 24]. De-duplication of very large datasets broadly proceeds by performing blocking, followed by pair-wise (or cluster-wide) similarity computation within each block. A large body of work has focused on the latter step of pair-wise similarity computation, known as matching [12, 21, 31]. Some other work [8, 14, 20] has considered fuzzy matching in the context of databases, however none of this work considers the problem of automatic blocking, drill-down, or rollup. Finally, we note that the structure of BlkTrees is akin to that of decision trees, a popular approach to classification; however, we note that the objectives of our BlkTrees are completely different, that of effectively trading off recall for efficiency in deduplication.

3 Preliminaries

3.1 Background and Notation

We use U to denote the set of entities (i.e., records) to be de-duplicated. Dividing U for pairwise comparisons is known as *blocking* (or *canopy formation*). The divided pieces are called *blocks* (or *canopies*). We use \mathcal{C} to denote the set of canopies, and C_i ’s denote the individual canopies. Formally, given a universe U , a set of canopies is given by a finite collect $\mathcal{C} = \{C_1, \dots, C_k\}$, $C_i \subseteq U$

and $\bigcup_i C_i = U$. A specific method to construct \mathcal{C} from U is called a *blocking function*. We start by restricting our attention to blocking functions that create a *disjoint* set of canopies (i.e., if $i \neq j$, then $(C_i \cap C_j) = \emptyset$) and then extend our results for *non-disjoint* sets of canopies (Section 7). Intuitively, a good blocking function must satisfy two desirable properties. First, canopy formation increases the efficiency of de-duplication by eliminating the need for performing pairwise comparisons between all pairs of entities in U . Second, the quality of de-duplication (i.e., recall of identified duplicates) must not be significantly reduced by performing fewer comparisons. Therefore, our goal is to find a set of canopies such that most duplicates in U fall within some canopy. We shall use $\mathcal{T}^+ \subset U \times U$ to denote a training dataset consisting of labeled duplicates in U , over which recall of blocking functions is measured. We shall construct blocking functions using a space \mathcal{H} of *hash functions* that partition U based on attributes of the entities in U ; each hash function assigns one hash value for each entity. For example, one hash function partitions U based on the first character of the titles of movies. A *conjunction* of hash functions h_1, \dots, h_l is equivalent to creating a single hash value by concatenating the hash values obtained by each h_i , effectively creating partitions (equivalence classes) where values of each of the hash functions matches. Typically \mathcal{H} is generated manually based on domain knowledge, and we shall present techniques to construct blocking functions using any \mathcal{H} . In addition, we shall also present techniques to automatically identify optimal hash functions for each attribute (Section 6).

3.2 Cost Model

While CBLOCK can be configured with any *cost model* for optimizing canopy formation, we use *latency* as the default cost model in our discussion.² The latency of any canopy formation is given by the total time it takes to perform all pairwise comparisons in each canopy.

In a grid environment (such as our de-duplication system implemented using map-reduce), pairwise comparisons in the set of canopies are performed in parallel. Given a canopy formation $\mathcal{C} = \{C_1, \dots, C_k\}$, with number of entities in canopy C_i denoted by s_i , the total num-

ber of pairwise comparisons being performed is $\sum_{i=1}^k s_i^2$. Motivated by de-duplication in a grid environment, we use the cost model $\text{cost}(\mathcal{C}) = \max_i s_i$. Clearly, in a truly elastic grid with a potentially infinite supply of machines, pairwise comparisons for each canopy are performed on a separate machine. Therefore, the latency is given by the largest canopy, justifying our cost model of using $\max_i s_i$.

When the number of machines on the grid are limited (and specifically, when there are fewer machines than canopies), we are faced with the problem of assigning canopies to machines. The following theorem shows that this assignment is NP-hard in general, based on a direct reduction from a scheduling problem. However, we also show that the latency using the largest canopy gives an upper bound on the best possible assignment.

Theorem 3.1 *Given a set $\mathcal{M} = \{M_1, \dots, M_m\}$ of m machines, a canopy formation $\mathcal{C} = \{C_1, \dots, C_k\}$ over N entities, $m < k$, any assignment $A : \mathcal{C} \rightarrow \mathcal{M}$ of canopies to machines has a cost given by $\text{cost}_A(\mathcal{C}) = \max_{j=1}^m (\sum_{C_i: A(C_i)=M_j} |C_i|^2)$. We have:*

1. *It is NP-hard to find an assignment that minimizes $\text{cost}_A(\mathcal{C})$.*
2. *For all assignments A , we have $\max_{i=1}^k |C_i|^2 \leq \text{cost}_A(\mathcal{C}) \leq (1 + \frac{k}{m}) \max_{i=1}^k |C_i|^2$. More specifically, let $X = \max(\max_{i=1}^k |C_i|^2, \frac{\sum_{i=1}^k |C_i|^2}{m})$. We have $X \leq \text{cost}_A(\mathcal{C}) \leq 2X$.*

Based on the theorem above, henceforth, we focus on the problem of finding best canopies that satisfy the constraint of $\max_i s_i \leq S$, for some given S .

4 Blocking Based on Labeled Data

This section addresses the problem of constructing disjoint blocking functions using a labeled dataset of positive examples. After formally defining the problem (Section 4.1), we introduce a tree-structured language for expressing blocking functions (Section 4.2). We then show that the general problem of finding an optimal blocking function is NP-hard (Section 4.3), and finally we present a greedy heuristic algorithm (Section 4.4) to find an approximate blocking function.

²All our algorithms and complexity results carry over for any “monotonic cost function”, i.e., $\text{cost}(\mathcal{C}) \leq \text{cost}(\mathcal{C}')$ whenever $\forall C \in \mathcal{C}, \exists C' \in \mathcal{C}'$ such that $C \subseteq C'$.

4.1 Problem Formulation

We formally define the problem of creating canopies given labeled data consisting of examples of duplicates (positive pairs). Recall the two conflicting goals of canopy formation: The more divisive a set of canopies is, the more likely it is to miss out on true duplicates. We formulate an optimization problem that trades off the two objectives of canopy formation, by associating a hard constraint on the maximum size of each canopy and maximizing the number of covered positive examples (recall) subject to this size constraint.

Definition 4.1 (Blocking Problem) *Given a labeled set \mathcal{T}^+ of positive examples, a space \mathcal{H} of hash functions, a size bound S on every canopy, and a size function $\text{size}()$ that returns the size of a canopy obtained by applying any conjunction of hash functions in \mathcal{H} on any input dataset I , construct a disjoint blocking function \mathcal{B} that partitions any input I into a set \mathcal{C} of disjoint canopies of size at most S , while maximizing the number of pairs from \mathcal{T}^+ that lie within canopies, i.e., maximizing: $\text{recall} = \frac{|\{(r_1, r_2) \in \mathcal{T}^+ \mid \exists c \in \mathcal{C}, r_1, r_2 \in c\}|}{|\mathcal{T}^+|}$.*

We make a few important observations about our problem definition. (1) As a reminder, we start by considering only disjoint blocking, and extend to non-disjoint blocking in Section 7. The next section describes a language to represent disjoint blocking functions (\mathcal{B}), and subsequently we give algorithms for finding \mathcal{B} . (2) We assume that there is a known size estimation function. In practice, some previous work on blocking [7] has used negative examples as an indirect way of incorporating size restrictions. Alternatively, previous work on estimating the cardinality of selection queries using histograms (refer [16]) can be used to estimate canopy sizes, as we shall see each canopy is obtained as a conjunction of hash functions. Of course, if the entire dataset were available during the construction of blocking predicates, it could be used for size computation. (In particular, exact size computation for the blocking technique we propose can be done in a few scans. Also, we shall see that our technique can be adaptively applied even in case of inaccurate size estimates.) (3) For this section we assume the existence of a space \mathcal{H} of hash functions. Most previous work has assumed the manual creation of such atomic hash functions. We also present in Section 6 an automated method of enumerating hash functions for each attribute. (4) Finally, we

assume the positive examples \mathcal{T}^+ are known; we describe the construction of this dataset in the experiments section (Section 8).

4.2 Blocking-Tree Space

This section presents a generic language for expressing disjoint blocking functions. We introduce a *hierarchical blocking tree* (called *BlkTree*), that partitions the entire set of entities in a hierarchical fashion by successively applying atomic hash functions from a known class \mathcal{H} . Formally:

Definition 4.2 *A BlkTree $B = (N, E, h)$ is composed of a tree with nodes N and edges E , and $h : N \rightarrow \mathcal{H}$ maps each node in the tree to a particular hash partitioning function from \mathcal{H} .*

Intuitively, each leaf node of the tree corresponds to a canopy. The BlkTree is built using the inputs described in Definition 4.1, namely the training data, a known space of atomic hash functions \mathcal{H} , and canopy-size estimates. Each node $n \in N$ in the tree corresponds to a set of entities from the entire set obtained by applying the hash functions from the root down to n . Each node n (with a size estimate exceeding the allowed maximum) then applies a particular partitioning hash function to create disjoint partitions of the set of entities corresponding to n .

At run-time, each entity is run through the BlkTree, and directed to the machine in the cluster based on the leaf node. (Note that in a distributed environment, the entire data itself is initially partitioned across multiple machines; therefore, the BlkTree is stored on every machine in order to redistribute the data based on the canopies.) Note that in practice the total number of large canopies created by any hash function on any node is a constant, for instance due to NULL values in the data, or a common default value for an attribute. Therefore, the size of the constructed BlkTree in terms of the number of nodes is small, so that the BlkTree fits in memory, and applying the BlkTree to an entity is efficient.

Example 4.3 *Figure 2 shows an example BlkTree for movie data with the root partitioning the movies lexicographically based on the title. This partition results in two large canopies—the node corresponding to NULL titles, and the node corresponding to titles that start with “T” (assume all titles have been capitalized in advance).*

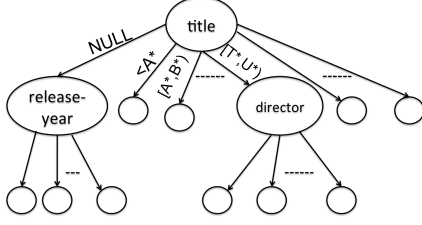


Figure 2: Example of a tree-structured disjoint blocking function.

In the NULL canopy a partition based on the release-year of the movie is performed, while the movies starting with “T” are partitioned by the name of the movie’s director. All leaf nodes in the resulting tree satisfy the maximum canopy-size requirement, and hence no further partitioning is performed.

4.2.1 Restricted languages

We note that BlkTrees are a very expressive language to describe disjoint blocking functions. In particular, the following natural languages are obtained as restrictions of BlkTrees:

1. **Single hashes:** Clearly single hash functions are equivalent to a BlkTree of height 1.
2. **Conjunctive functions (chains):** Conjunctions of hash functions are equivalent to restricting the width of BlkTrees to 1, i.e., a branching factor of 1. In particular, a conjunction $h_1 \wedge \dots \wedge h_k$ is equivalent to applying each hash function h_i in sequence to every single canopy, irrespective of whether a canopy is smaller than the required size S . Note that (disjunctions of) chains are the basic construct used in [7, 23], whereas our language of (disjunctions of) BlkTrees is significantly more expressive.
3. **Chain-tree:** Chain-trees are an extension of conjunctions where we are again specified a chain h_1, \dots, h_k of hash functions to be applied in sequence, however, subsequent hash functions are applied only if the canopy size exceeds the allowed maximum. In particular, chain-trees are obtained by restricting every level of BlkTrees to have the same hash function.

In our experiments, we implement algorithms for BlkTrees and all the restricted languages above and compare them in terms of recall, to observe a significantly higher recall using BlkTrees.

```

1: Input: Node  $n$  consisting of entities  $C_n$ , duplicates  $\mathcal{T}^+$ , space  $\mathcal{H}$  of hash
   functions, size bound  $S$ .
2: if  $|C_n| > S$  then
3:    $least = \infty$ ;  $best = \text{NULL}$ 
4:   for  $h \in \mathcal{H}$  do
5:     Compute  $e = \text{elim-count}(C_n, S, h)$ 
6:     if  $least > e$  then
7:        $least = e$ ;  $best = h$ 
8:     end if
9:   end for
10:  Set  $best$  as the hash for node  $n$ .
11:  Recurse on nodes resulting from  $best$  applied to  $n$ .
12: end if

```

Algorithm 1: Recursive greedy construction of BlkTree.

4.3 Intractability

Next we demonstrate that the general problem of finding the optimal BlkTree is NP-hard, and subsequently present a heuristic greedy algorithm.

Lemma 4.4 (BlkTree intractability) *Given a training set \mathcal{T}^+ with positive examples, a space \mathcal{H} of hash functions, and a bound S on the maximum size of any canopy, assuming $P \neq NP$, there does not exist any polynomial-time (in $\mathcal{T}^+, \mathcal{H}$) algorithm to find the optimal BlkTree.*

4.4 Greedy Algorithm

We propose a simple heuristic for constructing the BlkTree described in Algorithm 1. The general scheme of the algorithm is to locally pick the best hash function at every node in the tree, if the size (estimate) of the number of entities in this node is over the allowed maximum S . (If a particular hash function generates many large canopies, it is ignored, in order to maintain a small BlkTree. However, as described before, the number of large canopies is typically small; in our experiments over 140K movie entities, no hash function created more than a few large canopies.) The best hash function for a node is picked greedily by counting for all hash functions $h \in \mathcal{H}$, the number of duplicates that get eliminated on choosing the hash function h . The hash function that minimizes the number of eliminated duplicates is chosen. We describe three ways of counting the number of examples eliminated (function *elim-count* in Algorithm 1). Suppose a node n has P_n positive pairs, and application of h eliminates P_h duplicates and creates canopies C_1, \dots, C_k exceeding size S (among other canopies that are smaller than S). If the number of positive pairs in C_i is denoted $P(C_i)$, then the

three ways of counting the drop in the number of positive pairs are as follows:

- **Optimistic Count:** Intuitively, Algorithm 1 picks the hash function by assuming that no more duplicate examples would get eliminated, hence it is optimistic:

$$\text{Optimistic} = P_h$$

- **Pessimistic Count:** On application of a hash function h , we say that the number of duplicates that are eliminated include the ones broken by h as well as all examples that still remain in canopies larger than S :

$$\text{Pessimistic} = (P_h + \sum_{i=1..k} P(C_i))$$

- **Expected Count:** For the duplicates that still remain in large canopies after applying h , we compute an expected number of eliminated duplicates based on a random split of the canopy so as to obtain canopies of size S :

$$\text{Expected} = (P_h + \sum_{i=1..k} \frac{P(C_i)(n_i - 1)}{n_i})$$

where $n_i = \lceil \frac{|C_i|}{S} \rceil$. Effectively, a random split would only retain a $\frac{1}{n_i}$ fraction of the positive pairs, assuming each pair is independent.

Finally we note that an important feature of constructing the BlkTree is that it can be naturally adapted at runtime based on the actual canopy sizes, such as when the canopy-size estimates turned out to be inaccurate, or when available memory has reduced. Suppose while construction of the BlkTree a canopy-size bound of S (say, 5000 entities) was imposed, we may choose to construct the BlkTree based on a maximum canopy-size of a fraction of S (say, 1000 entities). Effectively, we will create a longer tree than necessary, and this “extra” portion of the tree may be used if any canopy needs to be split further based on the reasons described above. Conversely, if the actual canopy sizes turn out to be smaller than expected, we may choose to run through only a smaller part of the tree.

5 Rolling up small canopies

In this section, we introduce the problem of rolling up small canopies. The primary motivation for studying this problem is that a blocking function may unnecessarily have to create many small canopies, in order to make some of the larger canopies fit the required size bound. Therefore, as a post processing step, we can take the result of any blocking function, and combine multiple small canopies maintaining the size requirement yet increasing the overall recall.

We are given a set of canopies $\mathcal{C} = \{C_1, C_2, \dots, C_m\}$, where each canopy C_i has size (much) less than our canopy size limit S . We are also given a set of pairs of matching records $\mathcal{T}^+ = \{\dots, (r_{i_1}, r_{i_2}), \dots\}$. The rollup problem is to find a set of canopies $\mathcal{D} = \{D_1, D_2, \dots, D_\ell\}$ such that

- *Disjointness Constraint:* $\forall i, j, i \neq j, D_i \cap D_j = \emptyset$
- *Roll Up Constraint:* $\forall i, \exists i_1, i_2, \dots$, such that, $D_i = \cup_j C_{i_j}$
- *Maximum Size Constraint:* $\forall i, |D_i| \leq S$
- *Maximize Recall:* minimize the number of pairs of matching examples from \mathcal{T}^+ that are split across canopies.

Note that the rollup problem can be applied on any set of canopies generated using any previous blocking function. In particular, it can be applied on the BlkTree blocking function generated in Section 4. Each leaf of the BlkTree corresponds to a canopy, and by applying rollup, some leaves of the BlkTree get merged so as to maintain the size requirement but increase recall. Figure 3 shows an example blocking function obtained by performing rollup on the BlkTree in Figure 2. Note that although the resulting blocking function isn’t a tree, the resulting DAG can still be used for distributed canopy formation: Each entity starts at the root and traverses all the way down through the directed edges to a (possibly rolled-up) leaf node, which corresponds to a canopy.

We start by showing that the roll-up problem is intractable:

Lemma 5.1 (NP-completeness) *The rollup problem described above is NP-complete.*

Next we propose a greedy heuristic for the rollup problem that is inspired by Dantzig’s 2-approximation algorithm [10] for the knapsack problem. Conceptually,

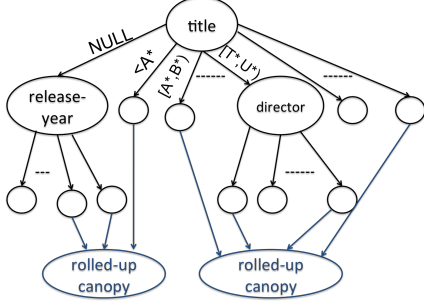


Figure 3: Rollup applied on canopies (leaf-nodes) generated in Fig. 2.

```

1: Input:  $\mathcal{C} = \{C_1, C_2, \dots, C_m\}$ , set of matching pairs  $\mathcal{T}^+$ , maximum canopy size  $k$ 
2: Set  $\mathcal{D} \leftarrow \mathcal{C}$  // initialize
3: repeat
4:   // Candidate pairs that can be merged
5:    $\mathcal{D}_{pair} \leftarrow \{(D_1, D_2) \mid |D_1| + |D_2| \leq k\}$ 
6:   if  $\mathcal{D}_{pair} \neq \emptyset$  then
7:      $(D_1^*, D_2^*) \leftarrow \arg \max_{\mathcal{D}_{pair}} \frac{benefit(D_1, D_2)}{\min(|D_1|, |D_2|)}$ 
8:     // Merge  $D_1^*$  and  $D_2^*$  into one canopy
9:      $\mathcal{D} \leftarrow \mathcal{D} \cup \{D_1^* \cup D_2^*\} - \{D_1^*, D_2^*\}$ 
10:   end if
11: until  $\mathcal{D}_{pair} = \emptyset$ 
12: Return  $\mathcal{D}$ 

```

Algorithm 2: Greedy Canopy Rollup Algorithm

our algorithm (Algorithm 2) starts with the initial set of canopies, and progresses in steps. In each step, the algorithm finds the pair of sets D_1, D_2 that together have less than S records, and maximize the following quantity:

$$benefit(D_1, D_2) / \min(|D_1|, |D_2|) \quad (1)$$

$benefit(D_1, D_2)$ is the number of matching pairs $(r_{i_1}, r_{i_2}) \in P$ such that $r_{i_1} \in D_1$ and $r_{i_2} \in D_2$. Intuitively, in each step we pick the canopy that has the smallest size but also puts a large number of matching pairs in the same canopy.

Algorithm 2 can be efficiently implemented in time linear in the number of matching pairs ($|\mathcal{T}^+|$) and quadratic in the number of input canopies ($|\mathcal{C}|$). Initially, we compute for each canopy $D \in \mathcal{C}$, one *merge candidate*. This is a canopy D' such that $|D'| \geq |D|$ and $|D| + |D'| \leq S$ such that $benefit(D, D')$ is maximum. This step takes $O(|\mathcal{T}^+| \cdot |\mathcal{C}|^2)$ time. In each step, we find the canopy D whose merge candidate has the maximum benefit to size ratio; we then merge D with its merge candidate. The new merge candidate for a canopy other than D and D' is

either $D \cup D'$ or its old merge candidate – this step takes $O(1)$ time for each canopy. The new merge candidate for $D \cup D'$ can be computed in $O(|\mathcal{T}^+| \cdot |\mathcal{C}|)$ time by considering all the other canopies and the positive examples. Since the algorithm terminates in at most $|\mathcal{C}|$ steps, our algorithm has $O(|\mathcal{T}^+| \cdot |\mathcal{C}|^2)$ time complexity.

6 Drill-Down Problem

In Section 4 we assumed a pre-existing and manually-generated space of hash functions (as is done in most previous work). Next we propose automatic (only using an attribute’s domain and labeled dataset) techniques for generating hash functions. Automatically constructed hash functions may be used to bootstrap the blocking methods, eliminating the need for a significant upfront manual effort. Moreover, even in the presence of an existing space of manually constructed hash functions, we can augment the space with (better) automatically generated hash functions.

We introduce the “drill-down” problem for a single attribute. Our goal is to optimally divide a single-attribute’s domain into disjoint sets so as to cover as many duplicate pairs as possible, but ensuring that the cost associated with any set is below a required threshold. First we formally define the partitioning of an attribute’s domain into *disjoint*, *covering*, *contiguous* subsets (called a *DCC partition*), then define the problem of finding an optimal DCC partition.

Definition 6.1 (DCC Partition) *Given a domain D with total ordering \prec , least element ‘start’ and greatest element ‘end’³, we say that a set \mathcal{I} is a DCC partition of D if $\forall I \in \mathcal{I} : I \subseteq D$ and all of the following hold:*

- **Disjoint:** $I_1, I_2 \in \mathcal{I}, I_1 \neq I_2 \Rightarrow I_1 \cap I_2 = \emptyset$
- **Contiguous subset:** *Every $I \in \mathcal{I}$ is of the form $[I^1, I^2]$, $[I^1, I^2)$, $(I^1, I^2]$, or (I^1, I^2) , $I^1 \preceq I^2$ and $I^1, I^2 \in D_A \cup \{\text{start}, \text{end}\}$*
- **Covering:** $D_A = \bigcup_{I \in \mathcal{I}} I$ □

Intuitively, a DCC partition completely divides D by “tiling” the entire domain. Also, note that the total ordering doesn’t need to correspond to the “natural ordering”

³The least and greatest element may be part of D in some cases (e.g., all 10-digit phone numbers) and not part of D in others (e.g., $-\infty$ and $+\infty$ for real numbers).

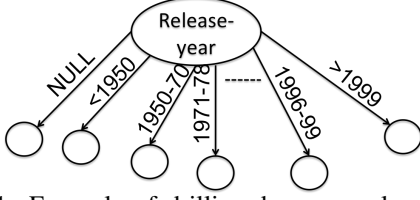


Figure 4: Example of drilling-down on release-year of movies.

such as lexicographic for strings or ‘<’ ordering for numeric. For instance, may choose to order director names by their last name and find a hash function, then also order them by first name and find another hash function.

Next we formally define the drill down problem.

Definition 6.2 (Drill Down Problem) Consider a single attribute A with an ordered domain $D_A, \prec, start, end$, a set of n duplication pairs $\mathcal{T}^+ = \{(a_1^1, a_1^2), \dots, (a_n^1, a_n^2)\}$, $\forall i : a_i^j \in D_A, a_i^1 \prec a_i^2$, and any monotonic black-box cost function⁴ $cost : I \subseteq D \rightarrow \mathbb{R}$, and a maximum cost bound S on any partition. Our goal is to find a DCC partition \mathcal{I} of D_A such that:

1. $\forall I \in \mathcal{I} : cost(I) \leq S$
2. Let $cov(\mathcal{I}, \mathcal{T}^+)$ be the number of duplicates covered: $cov(\mathcal{I}, \mathcal{T}^+) = \sum_{1 \leq i \leq n : \exists I \in \mathcal{I} \text{ with } [a_i^1, a_i^2] \subseteq I} 1$. For any DCC partition \mathcal{I}' satisfying (1) above, $cov(\mathcal{I}, \mathcal{T}^+) \geq cov(\mathcal{I}', \mathcal{T}^+)$. \square

Example 6.3 Figure 4 gives an example of a hash function that may be obtained using the drill down problem. This hash function may be added to the existing space of hash functions in consideration by a blocking function construction algorithm such as Algorithm 1.

Next we provide an optimal polynomial-time algorithm for the drill down problem based on dynamic programming. We use two core ideas in the algorithm described next. First, suppose we are finding the first partition in the given domain, the only “interesting endpoints” of a partition must be either a value at which a duplicate entity lies, or must be due to the boundary caused by the cost bound. Intuitively, we discretize the domain, and now only need

⁴ $\forall I, I' \subseteq D_A : I \subseteq I' \Rightarrow cost(I) \leq cost(I')$. Note that, in practice, for uniformly distributed data the cost function may simply bound the total size of the interval. But for skewed data, the size of the interval depends on the density of the data; therefore, we allow any arbitrary cost function.

to look at a finite number of endpoints in constructing the optimal partition; the space of possible DCC partitions still remains exponential. This observation is formalized below.

Lemma 6.4 (Interesting Endpoints) Given a domain D, \prec , with least and greatest elements $start, end$, with duplicate pairs $\mathcal{T}^+ = \{(a_1^1, a_1^2), \dots, (a_n^1, a_n^2)\}$, cost function I and a cost bound S , consider finding the first partition $[start, X)$ or $[start, X]$ (or open interval on $start$ if $start \notin D$) for the drill down problem. Let $Y \preceq end$ be the greatest value such that $cost([start, Y]) \leq S$, then there is an optimal drill down solution with $X \in (\{Y\} \cup \{a_i^j | a_i^j \preceq Y\})$.

The second observation is the optimal substructure property exploited by our dynamic programming algorithm. Given a domain $D, \prec, start, end$ over which we want to solve the drill down problem, the optimal solution for a sub-domain $D_s, \prec, start', end$, with $start' \succ start$, with the same cost function and cost bound is identical irrespective of the partitions chosen for $D - D_s$, i.e., from $start$ to $start'$. This property allows us to memoize the solutions for all sub-domains of known interesting endpoints, namely from a_i^j to end , for every a_i^j . We can then find an optimal solution to the entire domain by recursively considering sub-domains, as formalized below.

Lemma 6.5 (Optimal Substructure) Given a domain $D, \prec, start, end$ with duplicate pairs $\mathcal{T}^+ = \{(a_1^1, a_1^2), \dots, (a_n^1, a_n^2)\}$, cost function I , cost bound S , let Y be greatest value satisfying cost bound (as defined in Lemma 6.4). Let $V(I)$ be the total number of violations in the optimal solution for the subset of \mathcal{T}^+ with each endpoint in I . Then, $V(D)$ can be recursively computed as:

$$V([a, end]) = \min_{P \in (\{Y\} \cup \{a_i^j | a \prec a_i^j \preceq Y\})} (B([a, P]) + V([P, end]))$$

⁵ where $B([a, P])$ is the number of duplicate pairs broken due to the interval $B([a, P])$; i.e., $B([a, P]) = |\{i | a \preceq a_i^1 \preceq P \prec a_i^2\}|$.

The above lemma provides a natural dynamic programming algorithm (described in Algorithm 3), where we recursively solve the drill down problem for sub-domains,

⁵A similar expression for $B([a, P]) + V([P, end])$, which is omitted. We have a similar formula for every combination of open and closed interval, i.e., $[a, end)$, (a, end) , $(a, end]$.

```

1: Input:  $D = [a, b], \mathcal{T}^+$ , cost function:  $\text{cost}(\cdot)$ , cost bound  $S$ , memoized
   solutions  $M : D_s \rightarrow \text{NULL} \cup \mathbb{N}$ 
2: if  $M(D) \neq \text{NULL}$  then
3:   Return  $M(D)$ .
4: end if
5: if  $\mathcal{T}^+ = \emptyset$  then
6:    $M(D) = 0$ . Return  $M(D)$ .
7: end if
8: Compute max  $Y$ -value from  $a$  using  $\text{cost}(\cdot)$ ,  $S$  (Lemma 6.4).
9: Let  $\mathcal{Z} = (\{Y\} \cup \{a_i^j \in \mathcal{T}^+ | a \prec a_i^j \preceq Y\})$ 
10: Minimum value  $m = \infty$ , endpoint  $p_{\text{opt}} = \text{NULL}$ 
11: for  $P \in \mathcal{Z}$  do
12:   Compute  $B([a, P])$  using  $\mathcal{T}^+$  (Lemma 6.5)
13:    $\text{cand} = B([a, P]) + V([P, \text{end}]) // V([P, \text{end}])$  computed recur-
       sively
14:    $M([P, \text{end}]) = V([P, \text{end}])$ 
15:   if  $m > \text{cand}$  then
16:      $m = \text{cand}; p_{\text{opt}} = P$ 
17:   end if
18: end for
19:  $M(D) = m$ . Return  $M(D)$ 

```

Algorithm 3: Sketch of the dynamic programming algorithm with memoization to solve the drill down problem for a given domain D with a set of duplicate pairs \mathcal{T}^+ .

and memoize these solutions for future recursive calls in M ; initially, no solution is memoized. Algorithm 3 returns the total number of violated duplicate pairs but also tracks the specific endpoints. It can be seen easily that this algorithm runs in near-linear time and space based on the observation that the total number of different recursive calls is at most $\mathcal{O}(n)$: $\mathcal{O}(n)$ corresponding to all possible endpoints of duplicate pairs, and another $\mathcal{O}(n)$ corresponding to each maximum Y -value from Lemma 6.5 for each endpoint.

So far we have considered the drill down problem under the disjointness condition (recall Definition 6.1). We finally note that the drill down problem is trivial if we were allowed a non-disjoint set of intervals: We simply look at each duplicate pair (a_i^1, a_i^2) individually and create an interval $I_i = [a_i^1, a_i^2]$ if and only if $\text{cost}(I_i) \leq S$.

7 Non-Disjoint Canopies

In this section we consider the construction of a set of canopies that don't need to be disjoint. The first thing to note is that we need to revise our cost model from Section 3.2. We note that a cost function that only penalizes the size of the largest canopy doesn't suffice any longer: Given a set U of entities, we can create $\frac{|U|(|U|-1)}{2}$ canopies, with one canopy for each pair of entities in U . Note that this set of canopies has a maximum canopy

size of 2, and a perfect recall of 1. However, constructing a canopy for each pair is clearly prohibitive, as it incurs a large communication cost, i.e., each entity needs to be transferred to machines corresponding to $\mathcal{O}(|U|)$ canopies. Therefore, we introduce a cost metric that minimizes the combination of communication and computation cost. The cost of a set $\mathcal{C} = \{C_1, \dots, C_m\}$ is given by:

$$\text{cost}(\mathcal{C}) = \max_{1 \leq i \leq m} |C_i|^2 + \sum_{i=1}^m |C_i|$$

The computation cost, as before, is approximated by the computation for the largest canopy, where a complete pairwise comparison is performed. The communication is given by the total size of all canopies put together, which is roughly the number of entities that need to be transferred to different machines.

We address the problem of finding non-disjoint canopies as finding sets of canopies $\mathcal{C}_1, \mathcal{C}_2, \dots$, where each \mathcal{C}_i is a disjoint set of canopies. In a distributed environment, each \mathcal{C} can be performed in one map-reduce round. (Alternatively, if non-disjoint canopies are inherently supported, we may simply construct a single set $\bar{\mathcal{C}}$ of canopies as $\bar{\mathcal{C}} = \bigcup_i \mathcal{C}_i$.) When treating non-disjoint canopies as multiple rounds of disjoint canopies, once we bound the computation cost (i.e., the size of the largest canopy) in each round, our goal reduces to minimizing the number of rounds to obtain maximum recall with respect to a training dataset.

We present a generic algorithm (Algorithm 4) that extends any algorithm for disjoint canopy formation to an algorithm for the non-disjoint case. We assume a bound on the maximum computation in any round, and use the disjoint algorithm to maximize recall in a round. The duplicate pairs that are covered are then removed from the labeled dataset, and the next round is performed. We may truncate the algorithm when all pairs are covered, or no more pairs can be covered, or a pre-specified maximum number of rounds has reached.

8 Experiments

This section presents a detailed experimental study using two large commercial datasets at Yahoo: (1) a movie dataset consisting of 140K entities, and (2) a restaurants dataset consisting of 40K entities. We present a summary

```

1: Input: Labeled data  $\mathcal{T}^+$ , maximum canopy-size bound  $S$ , disjoint-algorithm
   ALGODISJ returning the covered pairs, (optional) bound on the number of
   rounds  $R$ .
2:  $numRds = 0$ ,  $change = true$ 
3: while  $(\mathcal{T}^+ \neq \emptyset) \wedge (change) \wedge (numRds < R)$  do
4:    $numRds = numRds + 1$ ;  $change = false$ 
5:    $COVERED = ALGODISJ(\mathcal{T}^+, S)$ 
6:   if  $COVERED \neq \emptyset$  then
7:      $\mathcal{T}^+ = \mathcal{T}^+ - COVERED$ 
8:      $change = true$ 
9:   end if
10: end while

```

Algorithm 4: Generic algorithm for performing non-disjoint canopy formation as multiple rounds of disjoint canopy formation.

of results based on both the datasets, but focus on movies for a more detailed evaluation. (We focus only on one dataset for a detailed evaluation due to space constraints; the movies dataset being larger makes for a more interesting study although trends are similar in the restaurants dataset.)

The primary goal of our study is to measure the effectiveness (increased recall) due to the more expressive BlkTree-based blocking, as compared to restrictions of BlkTrees. We measure recall for disjoint and non-disjoint versions of all our algorithms. In addition to the primary objectives described above, our experiments also understand the effects of increasing the size of canopies on recall, variation of recall with the number of disjuncts, effects of specific greedy strategies used, and understanding some basic properties of BlkTree-based blocking. Our experimental setup is described in Section 8.1 and results are presented in Section 8.2.

8.1 Experimental Setup

Dataset

We have applied CBLOCK on two commercial datasets from a search engine company: `movies` and `restaurants`. The primary movies dataset used in our experiments is a large database D_{movie} of 140K movies from Yahoo. In addition, we use a sample of movies from DBPedia [1] to obtain new duplicates, in addition to the duplicate already existing in D_{movie} . We constructed a labeled dataset \mathcal{T}^+ consisting of 1054 pairs of duplicates: Around 350 pairs of duplicates were obtained using manual labeling by paid editors. The remaining 704 pairs were obtained automatically

by finding common references to IMDb [2] movies; a small sample of 100 automatically generated pairs were checked manually to confirm that these were all duplicates. The schema of movies consisted of attributes `title`, `director`, `release year`, `runtime`, `genre` on which hash functions were created, and also other attributes (such as `genre` and `crew members`) that weren’t used for blocking. A sample of the space of hash functions used in our experiments is shown in Table 1.

The restaurants dataset used in our experiments consists of 40,000 restaurant records with attributes `name`, `street`, `city`, `state`, and `zip`. After de-duplication, there are 13,000 unique restaurant records. We use a labeled dataset of 4,674 duplicate pairs, and we used a similar set of hash functions as in Table 1.

Metrics

We evaluate our canopy generation algorithms using two metrics – *recall* and *computation cost*. Recall is measured as the fraction of matching pairs in \mathcal{T}^+ that appear within some canopy (Definition 4.1). Our algorithms are used to learn blocking hash functions, which are in turn applied to new data. We measure the computation cost in terms of the time taken to apply the hash function learnt by our algorithms on the dataset. Note that this is *not* the time taken to learn blocking functions. For non-disjoint canopy formation (Algorithm 4 in Section 7) we measure the increase in recall as the number of disjuncts (or map-reduce steps) is increased.

Algorithms

We describe our algorithms next. If any of our algorithms result in canopies C with size larger than our maximum size limit S , we further split it *randomly* into $\lceil \frac{|C|}{S} \rceil$ smaller parts. The algorithms we compare are

- **Random (R):** Each entity in \mathcal{U} is assigned uniformly at random to one of $\lceil \frac{|\mathcal{U}|}{S} \rceil$ canopies.
- **Single-Hash (SH):** Canopies are formed by picking a single hash function which maximizes recall.
- **Chain (C):** Canopies are formed by picking the best conjunction of hash function. (Note that this is the “size-aware” analogue of the approaches taken by previous work [7, 23] on using labeled data.)

Attributes	Hash function
All	(1) $h(x) = x$; (2) $h(x) = \text{prefix/suffix of length } K$; (3) $h(x) = \text{most frequent } K \text{ characters in alphanumeric order}$; ($K = 1, 3, 5$)
title	$h(x) = \text{longest token of } x$
year, runtime	number rounded to nearest to create k -point intervals, i.e., $h(x) = x - (x \bmod k)$
director	(1) $h(x) = \text{first-name of } x$; (2) $h(x) = \text{last-name of } x$

Table 1: Sample of the space of hash functions on movies used in our experiments.

- Chain-Tree (CT): A restriction to our tree based hash function where the same hash is used at each level. SH, C, CT were described earlier in Section 4.2.1.
- Hierarchical Blocking Tree (HBT): Our BlkTree-based canopy generation algorithm presented in Algorithm 1.

We also consider non-disjoint variants of all our algorithms; if $A \in \{\text{SH, C, CT, HBT}\}$ denotes one of the above algorithms, we use $A\text{-ND}$ to denote its non-disjoint variant (i.e., using A in Algorithm 4).

Setup

We perform 5-fold cross-validation for all runs of algorithms: We split \mathcal{T}^+ into 5 equal pieces randomly, then average over five runs with each run using 4 pieces of \mathcal{T}^+ as a training set to obtain the blocking function, then use the 5th piece as a test set. Since we don’t make any novel contribution on size estimation, our oracle *size()* computes the exact sizes of canopies based on the entire dataset. Our experiments were performed by varying the allowed maximum canopy size with $1K, 5K, 10K, 20K, 100K$ entities per canopy.

8.2 Results

We start by presenting detailed results on the movie dataset (Section 8.2.1–8.2.3). Finally, we present a brief summary of results on the restaurants dataset in Section 8.2.4.

8.2.1 Disjoint Canopies

Our first experiment was to compare the overall recall obtained by each of the algorithms—R, SH, C, CT, and HBT. Figure 5(a) shows the recall obtained by each of the algorithms on the movie dataset, varying the maximum allowed canopy size. (For each of the algorithms,

we picked the best of the optimistic, pessimistic, and expected greedy picking strategies.) The most important observation is that HBT achieves a significantly higher recall than C and SH, particularly when the maximum canopy size is lower. The reason for HBT’s higher recall is the greater expressive power of BlkTrees as a construct for describing disjoint blocking functions; BlkTree’s are able to apply a hash function at the first level that creates many good small canopies and a few large canopies, which are further split at subsequent levels of the tree. Another interesting observation from Figure 5(a) is that CT performs roughly as well as HBT, despite the slightly lower expressive power: Intuitively, the added power of HBT is effective when different nodes in the same level need different hash functions. Such a case would arise when different sections of the data have differing properties (e.g., US movies versus German movies); our dataset, however, only contained US movies. Finally, as expected R gives the lowest recall among all algorithms; henceforth, we omit R from the rest of our experiments.

To further understand the effects of the three greedy picking strategies—optimistic, pessimistic, and expected—described in Section 4.4, in Figure 5(b) we plot the recall for each of the algorithms by varying the greedy picking strategy. We note that in most cases all three algorithms perform very similarly, with the optimistic picking strategy slightly outperforming the others. The intuition for optimistic greedy strategy performing slightly better is that an optimistic estimate is better than an expected estimate since future levels of blocking are significantly better than a random split of each large canopy. Since optimistic is never worse than the other strategies, for the rest of our experiments we choose the optimistic strategy for each algorithm.

8.2.2 Non-disjoint Canopies

Next we consider the non-disjoint variants of each of the algorithms. Figure 5(c) shows the overall recall for each

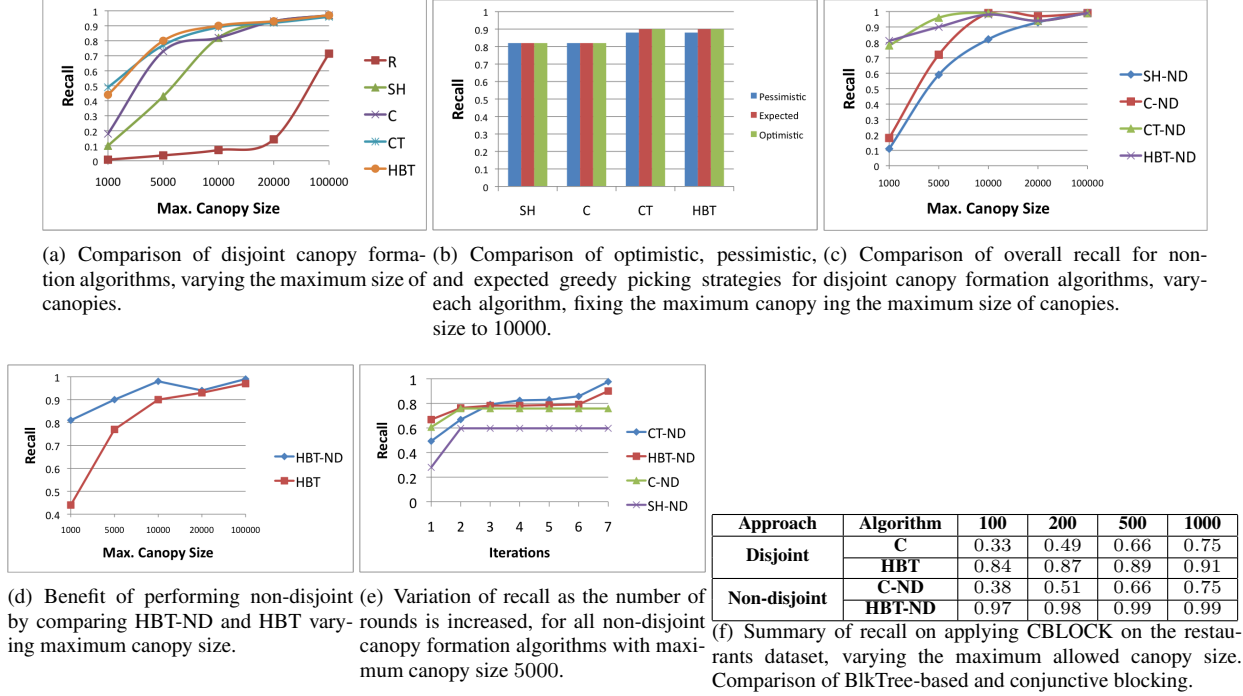


Figure 5: Experimental Results

non-disjoint algorithm as the number of canopies is varied. We notice, once again, that HBT-ND achieves a significantly higher recall than C-ND and SH-ND. In particular, C-ND is the size-aware analogue of previous state of the art [7, 23]. The reason for a much higher recall in HBT-ND is again the larger space of blocking functions BlkTrees can represent. Specifically, any conjunction that contains even one canopy larger than the maximum allowed is not permitted (or the conjunction needs to be further restricted losing more duplicate pairs). Note however that overall recall of CT-ND is very similar to that of HBT-ND; however, we shall see shortly that in the initial rounds of disjunction, HBT-ND increases recall slightly more rapidly than CT-ND.

A second observation on non-disjoint canopy formation is that the non-disjoint versions of each algorithm obtain higher recall than the corresponding disjoint versions. In Figure 5(d) we show the increase in recall obtained by HBT-ND as compared to HBT, for each maximum canopy size. Note that the additional benefit of non-disjointness

diminishes as the maximum canopy size is increased.

Next let us take a closer look at how the recall changes as the number of iterations is increased. To examine the difference between CT-ND and HBT-ND (as well as other non-disjoint algorithms), we plot the recall obtained after each round of disjoint canopy formation. Figure 5(e) plots the overall recall for the case of maximum canopy size 5000; we picked one fold of our cross-validation in which CT-ND ends with a slightly higher recall than HBT (therefore the apparent discrepancy with Figure 5(a)). First, note that for every iteration, HBT-ND is better than C-ND and SH-ND, which means that the number of positive examples covered increases more steeply for HBT-ND. Second, we see that HBT-ND obtains a higher recall than CT-ND initially, but CT-ND eventually ends with a slightly higher recall; in other words, with a limited number of map-reduce rounds, HBT-ND performs better than CT-ND. An optimal strategy of choosing a non-disjoint canopy formation by combining HBT-ND and CT-ND is left as future work.

Function	SH	CT	HBT
Time per record (ms)	8.8	17.7	7.1
Average tree height	1	2.8	1.34

Table 2: (1) Average running time (in $\mu s.$) of applying the best blocking function for each record. (2) Average length of the tree. All the numbers are for a max canopy size of 10,000.

8.2.3 Computation cost and Tree size

Computational Cost: We compared the computation cost (i.e., running) of applying a BlkTree against the cost of applying other blocking functions. The primary objective of investigating the running time is to establish the fact that BlkTrees do not add significant burden on the time required to apply the blocking function on an entire dataset. Table 2 shows the running time of applying the best blocking function (for maximum canopy size 10,000) for each of the algorithms (conjunctions being similar to applying a single hash function are omitted); these numbers are averaged over the $\sim 140K$ movie entities and over 5 repeated applications of the blocking function on the entire dataset. We note that applying each of the blocking functions requires a negligible amount of time (always under $20\mu s$ per record), and BlkTrees don’t add any discernible computational cost.

Tree Size: Table 2 also shows the height of the tree for CT and HBT (averaged over the 5 folds of cross-validation). It is noteworthy that HBT obtains similar recall with a shorter BlkTree than CT. This is because the BlkTree constructed using HBT is able to selectively create longer branches only when necessary. The longer tree for CT explains the higher blocking time per record.

8.2.4 Summary of Results for Restaurants

We present a very brief summary of our results on the restaurants dataset; restaurants displayed a similar general trend as movies, and a detailed study of restaurants is omitted due to space constraints. Table 5(f) presents the overall recall for HBT and HBT-ND compared against C and C-ND, varying the sizes of the maximum canopy: (1) We note that both the disjoint and non-disjoint versions of HBT significantly outperform the disjoint and non-disjoint versions of conjunctive blocking. (2) Fur-

ther, as with movies, the recall achieved by HBT is very high on restaurants, and very close to 1 with non-disjoint blocking even for small canopy sizes.

References

- [1] DBPedia. <http://dbpedia.org/>.
- [2] IMDb: The Internet Movie Database. <http://www.imdb.com>.
- [3] MAHOUT. <http://cwiki.apache.org/MAHOUT/canopy-clustering.html>.
- [4] R. Baxter, P. Christen, and T. Churches. A comparison of fast blocking methods for record linkage. In *KDD*, 2003.
- [5] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *VLDB Journal*, 18(1), 2009.
- [6] O. Benjelloun, H. Molina, H. Gong, H. Kawai, T. E. Larson, D. Menestrina, and S. Thavisomboon. D-Swoosh: A Family of Algorithms for Generic, Distributed Entity Resolution. *Technical Report, Stanford University*, 2007.
- [7] M. Bilenko, B. Kamath, and R. J. Mooney. Adaptive blocking: Learning to scale up record linkage and clustering. In *ICDM*, 2006.
- [8] S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *ICDE*, 2005.
- [9] P. Christen and T. C. Febrl. Freely extensible biomedical record linkage manual. <http://datamining.anu.edu.au/linkage.html>, 2, 2003.
- [10] G. B. Dantzig. Discrete-variable extremum problems. *Operations Research*, 5(2), 1957.
- [11] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE TKDE*, 2007.
- [12] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Society*, 64(2283), 1969.
- [13] L. Gu, R. Baxter, D. Vickers, and C. Rainsford. Record linkage: Current practice and future directions. *CSIRO Mathematical and Information Sciences, Tech. Report*, 2003.
- [14] S. Guha, N. Koudas, A. Marathe, and D. Srivastava. Merging the results of approximate match operations. In *VLDB*, 2004.
- [15] M. A. Hernandez and S. J. Stolfo. The merge/purge problem for large databases. In *SIGMOD*, 1995.
- [16] Y. E. Ioannidis. The history of histograms (abridged). In *VLDB*, 2003.
- [17] M. A. Jaro. Advances in record linkage methodology as applied to matching the 1985 census of tampa. *JASA*, 84, 1989.
- [18] L. Jin, C. Li, and S. Mehrotra. Efficient record linkage in large data sets. In *DASFAA*, 2003.
- [19] R. P. Kelley. Advances in record linkage methodology: a method for determining the best blocking strategy. *Record Linkage Techniques*, 1985.
- [20] N. Koudas, A. Marathe, and D. Srivastava. Flexible string matching against large databases in practice. In *VLDB*, 2004.
- [21] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: Similarity measures and algorithms. Tutorial at SIGMOD, 2006.

- [22] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *KDD*, 2000.
- [23] M. Michelson and C. A. Knoblock. Learning blocking schemes for record linkage. In *AAAI*, 2006.
- [24] H. B. Newcombe, J. M. Kennedy, S. J. Axford, and A. P. James. Automatic linkage of vital records. *Science*, 130(3381), 1959.
- [25] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*, 2011.
- [26] S. Rendle and L. Schmidt-Thieme. Scaling record linkage to non-uniform distributed class sizes. In *PAKDD*, 2008.
- [27] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *SIGKDD*, 2002.
- [28] H. sik Kim and D. Lee. Parallel linkage. In *CIKM*, 2007.
- [29] S. Tejada, C. A. Knoblock, and S. Minton. Learning object identification rules for information integration. *Information Systems Journal*, 26(8), 2001.
- [30] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, 2010.
- [31] S. E. F. W. Cohen, P. Ravikumar. A comparison of string distance metrics for name-matching tasks. In *Proc. of IJCAI*, 2003.
- [32] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *SIGMOD*, 2009.
- [33] W. Winkler. Overview of record linkage and current research directions. *Statistical Research Division, U.S. Bureau of the Census, Tech. Report*, 2006.